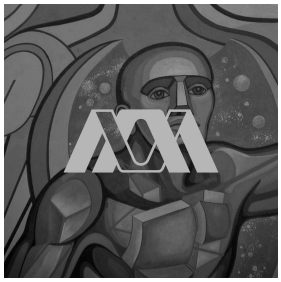
Algoritmos fundamentales

Arturo García Flores

Universidad Autónoma Metropolitana–Iztapalapa



7 de mayo de 2022

Durante el desarrollo de un programa de computadora se implementa algún método ideado de forma previa con el objetivo de resolver un problema. Este método es independiente del lenguaje de programación que se emplea y quien determina los pasos a seguir para resolver el problema, incluso más que el programa de computadora en sí mismo. En computación, el término algoritmo se refiere a un método finito y eficaz de resolución de problemas. Por ello, en este escrito presento a los algoritmos como el objeto central de estudio de la informática.

1. Introducción

La palabra algoritmo se refiere a un método paso a paso para resolver un problema en lenguaje natural o escribiendo un programa de computadora que implemente el procedimiento [1]. Por ejemplo, el algoritmo de Euclides es un método para encontrar el máximo común divisor de dos números A y B (figura 1a). Este algoritmo se puede transcribir en un lenguaje de programación para ser ejecutado en un ordenador (figura 1b) [2,3].

La idea de un algoritmo informático se atribuye a Ada Augusta, quien amplió los estudios de Babbage sobre cómo funcionaría una «máquina analítica», reconociendo que su importancia radica *en la posibilidad de utilizar una secuencia dada de instrucciones de forma repetida, siendo el número de veces preasignado o dependiente de los resultados del cálculo* [4]. Esta es la esencia de un algoritmo informático.

La mayoría de los algoritmos de interés involucran la organización de los datos involucrados durante el cómputo. Tal organización conduce a estructuras de datos que surgen como productos finales de los algoritmos y que, por tanto, debemos estudiar para comprender los algoritmos [5].

Cabe mencionar que los algoritmos no son una herramienta exclusiva de las matemáticas o de la computación. Nuestra vida cotidiana se rodea de diversos casos de algoritmos, como los pasos a seguir para preparar un guisado, armar un mueble o llenar un formulario.

```
Calcula el máximo común divisor de dos
    enteros no negativos A y B de la forma
    siguiente:
Si B es 0, la respuesta es A. Si no, divide
    A entre B y toma el resto R. La
    respuesta es el máximo común divisor de
    B y R.
```

(a) Algoritmo de Euclides en lenguaje natural (español)

```
int mcd(int a, int b){
    if (b == 0)
    return a;
    int r = a % b;
    return mcd(b,r);
}
```

(b) Algoritmo de Euclides en lenguaje de programación (C++)

Figura 1: El algoritmo de Euclides es un conjunto de instrucciones definidas, ordenadas y acotadas para resolver un problema: hallar el máximo común divisor entre dos números enteros positivos.

2. Pseudocódigo

Antes de desarrollar un programa para resolver un problema en particular, se debe contar con una comprensión completa del problema y un enfoque de solución cuidadosamente planificado. Para ello sirve un pseudocódigo.

El pseudocódigo es un lenguaje artificial e informal con el cual se representa a un algoritmo. Es similar al español u otro idioma: conveniente y fácil de usar, aunque no es un lenguaje de programación. Por lo general, posee una sintaxis propia para representar a un algoritmo [6, 7]. Algunas reglas semánticas y sintácticas se pueden discernir a partir del pseudocódigo del algoritmo de Euclides (figura 2):

1. Está limitado por las etiquetas `INICIO` y `FIN`, las instrucciones se escriben en mayúsculas, las variables en minúsculas y los mensajes en forma de oración

2. La lectura de datos se representa con la etiqueta `LEER`; la escritura de datos, con la etiqueta `ESCRIBIR`.

3. La declaración de variables la define un identificador (`a` y `b`) seguido de dos puntos y el tipo de dato (`ENTERO`).

4. La declaración de una función (`FUNC`) se constituye de un identificador que corresponde con el nombre de la fuinción (`mcd()`).

5. Las estructuras de control de flujo se declaran de distintas formas según sea el caso. El pseudocófigo de la figura 2 emplea una estructura de control condicional `SI-DE LO CONTRARIO`.

```
INICIO
    ESCRIBIR "Ingresa los valores de A y B"
    LEER a, b: ENTERO
    FUNC mcd()
        SI b == 0 ENTONCES
            ESCRIBIR "El MCD es:" a
        FIN SI
        DE LO CONTRARIO
            r == a % b
            ESCRIBIR "El MCD es:" mcd(b,r)
        FIN DE LO CONTRARIO
    FIN FUNC
FIN
```

Figura 2: De acuerdo con el pseudocódigo, si la expresión lógica se cumple, se ejecutan las instrucciones del bloque `SI`; si no se cumple, se ejecutan las instrucciones del bloque `DE LO CONTRARIO`.

3. Algoritmos

Los algoritmos nos brindan el potencial de obtener grandes ahorros, incluso hasta el punto de permitirnos realizar tareas que de otro modo serían imposibles. Por esta razón, el diseño cuidadoso del algoritmo es una parte en extremo efectiva del proceso de resolución de un gran problema [8].

La elección del mejor algoritmo puede ser un proceso complicado que quizás implique un análisis matemático sofisticado. Por otro lado, no debemos usar un algoritmo sin tener idea de qué recursos podría consumir. Por ello, repasaremos los algoritmos más importantes en informática: algoritmos de búsqueda, de ordenamiento y voraces [9].

3.1. Algoritmos de búsqueda

El desarrollo de la tecnología, en particular de la informática moderna, han hecho accesible una gran cantidad de información. Esto implica que la capacidad de buscar de forma eficiente a través de esta información sea fundamental para procesarla.

Los algoritmos de búsqueda han demostrado ser efectivos en numerosas aplicaciones durante décadas. Sin este tipo de algoritmos, el desarrollo científico y tecnológico que disfrutamos cada día no sería posible [10, 11].

3.1.1. Tablas de símbolos

El término tabla de símbolos se emplea para describir un mecanismo donde se almacena información (un *valor*) que luego se puede buscar y recuperar especificando una *clave*. La naturaleza de las claves y los valores depende de la aplicación. Existe una gran cantidad de claves y de información, por lo que implementar una tabla de símbolos eficiente es un desafío computacional significativo [12, 13].

La búsqueda es muy importante en la mayor parte de las aplicaciones informáticas, por lo que las tablas de símbolos están disponibles como abstracciones de alto nivel en muchos entornos de programación. La tabla 1 proporciona algunos ejemplos de claves y valores que se emplean en aplicaciones típicas.

3.1.2. Búsqueda lineal

Una búsqueda lineal es el método más simple de buscar un conjunto de datos. Desde el inicio, cada elemento de este conjunto se examina hasta que se hace una coincidencia. Una vez que se encuentra el elemento, la búsqueda finaliza.

No podemos discutir formas eficientes de encontrar un elemento en un conjunto sin considerar cómo se insertaron los elementos en la lista. Por tanto, la discusión sobre los algoritmos de búsqueda está relacionada con el tema de la operación de inserción del conjunto. Supongamos que queremos insertar elementos lo más rápido posible y no nos preocupa tanto el tiempo que lleva encontrarlos. Pondríamos el elemento en el último espacio de un conjunto basado en arreglos y en el primer espacio de un conjunto enlazado. El conjunto resultante se ordena según el momento de la inserción, no según el valor de la clave [14–16].

Un algoritmo escrito para una búsqueda lineal puede ser el siguiente:

1. Escribe una función para buscar un elemento `x` dado en un arreglo `arr[]` de `n` elementos.
2. Inicia la búsqueda desde el extremo izquierdo del arreglo `arr[]` y compara `x` con cada elemento de `arr[]`.
3. Devuelve el índice, si `x` coincide con un elemento.
4. Devuelve -1, si `x` no coincide con un elemento.

El algoritmo anterior se muestra en lenguaje de programación en la figura 3.

```
int buscar(int arr[], int n, int x){
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

int main(void){
    int arr[] = {2,4,6,8,10};
    int x = 2;
    int n = sizeof(arr) / sizeof(arr[0]);

    int r = buscar(arr, n, x);
    (r == -1)
        ? cout << x << "␣no␣se␣encuentra␣en␣
          el␣arreglo."
        : cout << x << "␣se␣encuentra␣en␣el␣
          lugar␣" << r << "␣del␣arreglo.";
    return 0;
}
```

(a) Algoritmo de búsqueda lineal en lenguaje de programación (C++)

```
2 se encuentra en el lugar 0 del arreglo.
```

(b) Mensaje de salida

Figura 3: La búsqueda lineal es un algoritmo de búsqueda secuencial donde se inicia desde un extremo y se verifica cada elemento de un conjunto hasta encontrar el elemento deseado.

La búsqueda lineal rara vez se usa en la práctica, debido a que otros algoritmos de búsqueda, como el algoritmo de búsqueda binaria, permiten una comparación de búsqueda más rápida. Sin embargo, la búsqueda lineal tiene la ventaja de que funcionará con cualquier conjunto de datos (ordenado o desordenado).

3.1.3. Búsqueda binaria

La búsqueda binaria es un algoritmo de búsqueda que se usa en arreglos ordenados, dividiendo de forma rápida el intervalo de búsqueda por la mitad [17].

Si un conjunto de datos se ordena y almacena secuencialmente en un arreglo, podemos usar una búsqueda binaria. El algoritmo de búsqueda binaria mejora la eficiencia de la búsqueda al limitar la búsqueda al área donde podría estar el elemento. El algoritmo de búsqueda binaria recorta continuamente el área de búsqueda hasta que se encuentra el elemento o el área de búsqueda desaparece (el elemento no está en la lista) [18, 19].

Tabla 1: Una tabla de símbolos es una estructura de datos para pares clave-valor que admite dos operaciones: *insertar* (poner) un nuevo par en la tabla y *buscar* (obtener) el valor asociado con una clave determinada.

Aplicación	Propósito de la búsqueda	Clave	Valor
diccionario	buscar definición	palabra	definición
índice de libros	encontrar páginas relevantes	término	lista de números de página
compartir archivos	buscar canciones para descargar	nombre de la canción	ID de la computadora
administración de cuentas	procesar transacciones	número de cuenta	detalles de la transacción
búsqueda web	encontrar páginas web relevantes	palabra clave	lista de nombres de páginas
compilador	buscar tipo y valor	nombre de variable	tipo y valor

Un algoritmo escrito para una búsqueda lineal puede ser el siguiente:

1. Compara x con el elemento del medio.
2. Devuelve el índice medio, si x coincide con el elemento medio.
3. De lo contrario, si x es mayor que el elemento medio, entonces x solo puede estar a la derecha del elemento medio. Entonces recurrimos a la mitad derecha.
4. De lo contrario (si x es menor que el elemento medio) se repite para la mitad izquierda.

El algoritmo anterior se muestra en lenguaje de programación en la figura 4.

```
int buscar(int arr[], int l, int n, int x){
  if (n >= l) {
    int mid = l + (n - l) / 2;
    if (arr[mid] == x)
      return mid;
    if (arr[mid] > x)
      return buscar(arr, l, mid - 1, x);
    return buscar(arr, mid + 1, n, x);
  }

  return -1;
}

int main(void){
  int arr[] = {2,4,6,8,10};
  int x = 2;
  int n = sizeof(arr) / sizeof(arr[0]);
  int r = buscar(arr, 0, n - 1, x);
  (r == -1)
        ? cout << x << "␣no␣se␣encuentra␣en␣
            el␣arreglo."
        : cout << x << "␣se␣encuentra␣en␣el␣
            lugar␣" << r << "␣del␣arreglo.";
  return 0;
}
```

(a) Algoritmo de búsqueda binaria en lenguaje de programación (C++)

```
2 se encuentra en el lugar 0 del arreglo.
```

(b) Mensaje de salida

Figura 4: La búsqueda binaria funciona de manera eficiente en listas ordenadas. Por lo tanto, para buscar un elemento en algún conjunto con esta técnica, se debe asegurar que la lista esté ordenada.

Cabe mencionar que no se garantiza que la búsqueda binaria sea más rápida para buscar en listas muy pequeñas. Se debe tener en cuenta que aunque la búsqueda binaria, por lo general, requiere menos comparaciones, cada comparación implica más evaluaciones.

3.2. Algoritmos de ordenamiento

El ordenamiento es el proceso de reorganizar una secuencia de objetos para ponerlos en algún orden lógico. La ubicuidad del uso de la computadora nos ha inundado de datos y, a menudo, el primer paso para organizar los datos es ordenarlos. Todos los sistemas informáticos tienen implementaciones de algoritmos de ordenamiento, para uso del sistema y de los usuarios. Por ejemplo, un teléfono móvil muestra los mensajes de texto ordenados por fecha; es probable que un algoritmo de ordenamiento los haya puesto en ese orden [20–22].

Los algoritmos de ordenamiento juegan un papel importante en el procesamiento de datos comerciales y en el cómputo científico de alto rendimiento. De hecho, un algoritmo de ordenamiento (*quicksort*) fue catalogado como uno de los diez mejores algoritmos para la ciencia y la tecnología del siglo XX [23].

3.2.1. Algoritmos de burbuja

Los algoritmos de burbuja son un algoritmo de clasificación simple y lento que recorre un conjunto de objetos en distintas ocasiones, compara cada par de elementos adyacentes y los intercambia si están en el orden incorrecto [24].

La idea básica de un algoritmo de burbuja se describe a continuación: [25–27]

1. Los elementos de datos se agrupan en dos secciones: una sección ordenada y una sección no ordenada.
2. Se revisa cada elemento en la sección no ordenada y se reorganiza su posición con su vecino para colocar el elemento con mayor orden en la posición más alta.
3. El elemento con el orden más alto estará en la parte superior de la sección no ordenada y se moverá al final de la sección ordenada.
4. Se repite el paso 2 hasta que no queden más elementos en la sección sin clasificar.

Un algoritmo escrito para un ordenamiento de burbuja puede ser el siguiente:

Paso 1: El algoritmo compara los dos primeros elementos de un conjunto de números e inicia el ordenamiento.

- (**51**428) → (**15**428) × Intercambia desde 5 > 1
- (**15**428) → (1**45**28) × Intercambia desde 5 > 4
- (14**52**8) → (14**25**8) × Intercambia desde 5 > 2
- (142**58**) → (142**85**) ✓

Paso 2: Dado que estos elementos (5 y 1) ya están en orden (8 > 5), el algoritmo no los intercambia y vuelve a iniciar con los dos primeros elementos siguientes.

- (**14**258) → (**14**258) ✓
- (**14**258) → (1**24**58) × Intercambia desde 4 > 2
- (12**45**8) → (12**45**8) ✓
- (12**458**) → (12**458**) ✓

Paso 3: El conjunto de números ya está ordenado; sin embargo, el algoritmo, lleva a cabo un pase completo sin intercambios para verificar que el conjunto está ordenado.

- (**12**458) → (**12**458) ✓
- (**12**458) → (**12**458) ✓
- (12**45**8) → (12**45**8) ✓
- (12**458**) → (12**458**) ✓

El algoritmo anterior se muestra en lenguaje de programación en la figura 5.

```
void burbuja(int arr[], int n){
  int i, j;
  for (i = 0; i < n - 1; i++)
    for (j = 0; j < n - i - 1; j++)
      if (arr[j] > arr[j + 1])
        swap(arr[j], arr[j + 1]);
}

void imprimir(int arr[], int size){
  int i;
  for (i = 0; i < size; i++)
    cout << arr[i] << "␣";
      cout << endl;
}

int main(){
  int arr[] = {5,1,4,2,8};
  int n = sizeof(arr) / sizeof(arr[0]);
  burbuja(arr, n);
  cout << "Ordenamiento␣de␣burbuja:␣";
  imprimir(arr, n);
  return 0;
}
```

(a) Algoritmo de burbuja en lenguaje de programación (C++)

```
Ordenamiento de burbuja: 1 2 4 5 8
```

(b) Mensaje de salida

Figura 5: El ordenamiento de burbuja pone un conjunto de elementos en orden creciente comparando de manera sucesiva elementos adyacentes e intercambiándolos si están en orden incorrecto.

Como se observa, en un algoritmo de burbuja el movimiento de los elementos con órdenes más altos, son como burbujas en el agua, flotando de m¿forma lenta de abajo hacia arriba.

3.2.2. Algoritmos por inserción

El algoritmo que se usa a menudo para clasificar las manos del *bridge* es considerar las cartas una por una, insertando cada una en un lugar adecuado entre las ya consideradas (manteniéndolas ordenadas). En una implementación por computadora, necesitamos hacer espacio para insertar el elemento actual moviendo los elementos más grandes una posición hacia la derecha, antes de insertar el elemento actual en la posición vacante. Este proceso se conoce como algoritmo por inserción [28].

En este tipo de algoritmos, los elementos a la izquierda de un conjunto de elementos están ordenados, pero no se encuentran en su posición final, ya que es posible que deban moverse para dejar espacio a los elementos más pequeños que se encuentren más adelante. Sin embargo, el conjunto está ordenado por completo cuando el índice alcanza el extremo derecho.

La idea básica de un algoritmo de ordenamiento por inserción se describe a continuación: [29, 30]

1. Los elementos de datos se agrupan en dos secciones: una sección ordenada y una sección no ordenada.
2. Se toma un elemento de la sección sin clasificar.
3. Se inserta el elemento en la sección ordenada en la posición correcta según la propiedad comparable.
4. Los pasos 2 y 3 se repiten hasta que no queden más elementos en la sección sin clasificar.

Un algoritmo escrito para un ordenamiento por inserción puede ser el siguiente:

Paso 1: Compara el segundo elemento de un conjunto con su predecesor. Si es más pequeño que su predecesor, desplázalo hacia su izquierda.

Paso 2: Compara el elemento siguiente con sus predecesores. Si es más pequeño que su predecesores, desplázalo hacia la izquierda de su sucesor.

- 4 3 2 10 12 1 5 6 ×
  El 3 se desplaza a la izquierda del 4
- 3 4 2 10 12 1 5 6 ×
  El 2 se desplaza a la izquierda del 3
- 2 3 4 10 12 1 5 6 ✓
- 2 3 4 10 12 1 5 6 ✓
- 2 3 4 10 12 1 5 6 ×
  El 1 se desplaza a la izquierda del 2
- 1 2 3 4 10 12 5 6 ×
  El 5 se desplaza a la izquierda del 10
- 1 2 3 4 5 10 12 6 ×
  El 6 se desplaza a la izquierda del 10
- 1 2 3 4 5 6 10 12 ✓

El algoritmo anterior se muestra en lenguaje de programación en la figura 6.

```cpp
void insercion(int arr[], int n){
  int i, elemento, j;
  for (i = 1; i < n; i++){
    elemento = arr[i];
    j = i - 1;

    while (j >= 0 && arr[j] > elemento){
      arr[j + 1] = arr[j];
      j = j - 1;
    }
    arr[j + 1] = elemento;
  }
}

void imprimir(int arr[], int n){
  int i;
  for (i = 0; i < n; i++)
    cout << arr[i] << "␣";
        cout << endl;
}

int main(){
  int arr[] = {4,3,2,10,12,1,5,6};
  int n = sizeof(arr) / sizeof(arr[0]);

  insercion(arr, n);
  cout << "Ordenamiento␣por␣inserción:␣";
  imprimir(arr, n);

  return 0;
}
```

(a) Algoritmo de burbuja en lenguaje de programación (C++)

```
Ordenamiento por inserción: 1 2 3 4 5 6 10 12
```

(b) Mensaje de salida

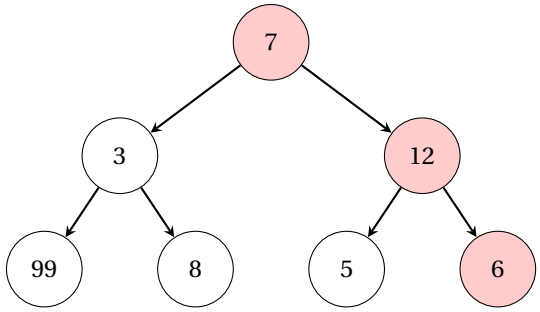
Figura 6: El ordenamiento por inserción es un algoritmo de clasificación simple, aunque no es el más eficiente. Para ordenar una lista con n elementos, el ordenamiento por inserción inicia con el segundo elemento.

La idea de los algoritmos por inserción proviene de nuestras experiencias de la vida diaria. Por ejemplo, cuando jugamos cartas, insertaremos la carta siguiente que elijamos en las cartas que ya están ordenadas en nuestra mano.

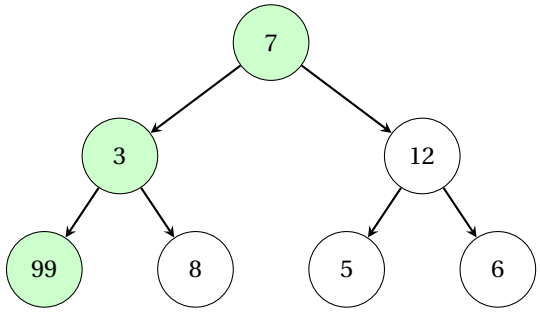
3.3. Algoritmos voraces

Un algoritmo voraz es un proceso simple e intuitivo. Se utiliza en problemas de optimización. El algoritmo hace la elección óptima en cada paso mientras intenta encontrar la forma óptima general de resolver todo el problema. Los algoritmos voraces tienen bastante éxito en algunos problemas, como la codificación de Huffman, que se usa para comprimir datos, o el algoritmo de Dijkstra, que se usa para hallar el camino más corto a través de un gráfico [31,32].

En muchos problemas, una estrategia voraz no produce una solución óptima. Por ejemplo, en la figura 7, el algoritmo busca el camino con la suma más grande, seleccionando el mayor número disponible en cada paso. Sin embargo, no logra encontrar la suma más grande porque toma decisiones basadas solo en la información que tiene en cualquier paso, sin tener en cuenta el problema general.



(a) Opción inadecuada



(b) Opción adecuada

Figura 7: Con el objetivo de alcanzar la suma más grande, en cada paso, el algoritmo elige lo que parece ser la opción inmediata óptima, por lo que elegirá 12 en lugar de 3 en el segundo paso y no llegará a la mejor solución, que contiene 99.

Por otro lado, si las propiedades a continuación son verdaderas, se puede usar un algoritmo voraz para resolver el problema [33].

1. Elección voraz. Se puede alcanzar una solución óptima global eligiendo la opción óptima en cada paso.

2. Subestructura óptima. Un problema tiene una subestructura óptima si una solución óptima para todo el problema contiene las soluciones óptimas para los subproblemas.

En otras palabras, los algoritmos voraces trabajan en problemas para los que es cierto que, en cada paso, hay una elección que es óptima para el problema hasta ese paso, y después del último paso, el algoritmo produce la solución óptima del problema completo.

3.3.1. Algoritmo de Huffman

El algoritmo de Huffman es un ejemplo donde el enfoque voraz tiene éxito. Este algoritmo analiza un mensaje y, dependiendo de las frecuencias de los caracteres utilizados en el mensaje, asigna una codificación de longitud variable para cada objeto. Un objeto de uso más común tendrá una codificación más corta, mientras que un objeto raro tendrá una codificación más larga [34].

El algoritmo de codificación de Huffman toma información sobre las frecuencias o probabilidades de que ocurra un objeto en particular. Comienza a construir el árbol de prefijos de abajo hacia arriba, comenzando con los dos objetos menos probables de la lista. Toma esos objetos y forma un subárbol que los contiene, y luego elimina los objetos individuales de la lista. El algoritmo suma las probabilidades

de los elementos en un subárbol y agrega el subárbol y su probabilidad a la lista. A continuación, el algoritmo busca en la lista y selecciona los dos objetos o subárboles con las probabilidades más bajas. Los usa para crear un nuevo subárbol, elimina los subárboles/objetos originales de la lista y luego agrega el nuevo subárbol y su probabilidad combinada a la lista. Esto se repite hasta que haya un árbol y se hayan agregado todos los elementos. En cada subárbol, se crea la codificación óptima para cada objeto y juntos componen la codificación óptima general [35, 36].

Un algoritmo voraz para un construir un «árbol de Huffman» puede ser el siguiente:

1. Crea un nodo de hoja para cada carácter único y una pila mínima de todos los nodos de hoja [1].
2. Extrae dos nodos con la frecuencia mínima de la pila mínima.
3. Crea un nuevo nodo interno con una frecuencia igual a la suma de las frecuencias de los dos nodos. Haz que el primer nodo extraído sea su hijo izquierdo y que el otro nodo extraído sea su hijo derecho. Agrega este nodo a la pila mínima.
4. Repite los pasos 2 y 3 hasta que la pila contenga solo un nodo. El nodo restante es el nodo raíz y el árbol está completo.

Estudiemos el algoritmo con un ejemplo. Crea una pila mínima que contenga seis nodos, donde cada nodo represente la raíz de un árbol con un solo nodo.

Carácter	a	b	c	d	e	f
Frecuencia	5	9	12	13	16	45

Extrae dos nodos de frecuencia mínima de la pila mínima y agrega un nuevo nodo interno (NI) con frecuencia 5+9 = 14 (figura 8a).

Carácter	c	d	NI	e	f
Frecuencia	12	13	14	16	45

Extrae dos nodos de frecuencia mínima y agrega un nuevo nodo interno con frecuencia 12 + 13 = 25 (figura 8b).

Carácter	NI	e	NI	f
Frecuencia	14	16	25	45

Extrae dos nodos de frecuencia mínima y agrega un nuevo nodo interno con frecuencia 14 + 16 = 30 (figura 8c).

Carácter	NI	NI	f
Frecuencia	25	30	45

Extrae dos nodos de frecuencia mínima y agrega un nuevo nodo interno con frecuencia 25 + 30 = 55 (figura 8d).

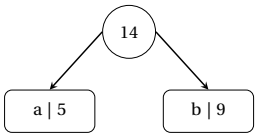
[1] La pila mínima se usa como una cola de prioridad. El valor del campo de frecuencia se usa para comparar dos nodos en la pila mínima. Inicialmente, el carácter menos frecuente está en raíz.

Carácter	f	NI
Frecuencia	45	55

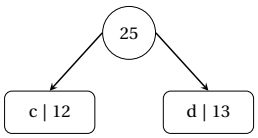
Extrae dos nodos de frecuencia mínima y agrega un nuevo nodo interno con frecuencia 45 + 55 = 100 (figura 8e).

Carácter	NI
Frecuencia	100

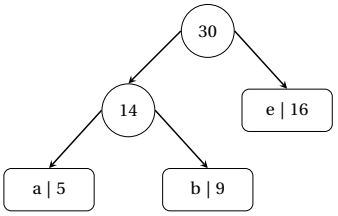
Dado que la pila contiene solo un nodo, el algoritmo se detiene aquí (figura 8).



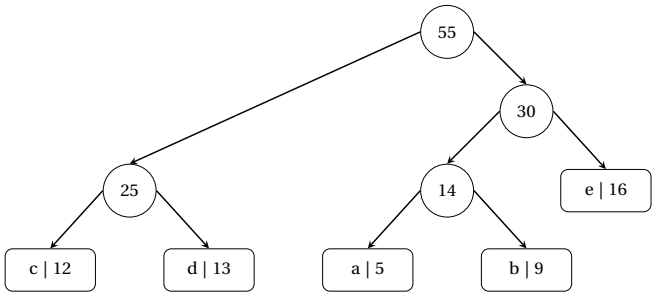
(a) La pila mínima contiene cinco nodos, donde cuatro nodos son raíces de árboles con un solo elemento cada uno, y un nodo de pila es la raíz del árbol con tres elementos.



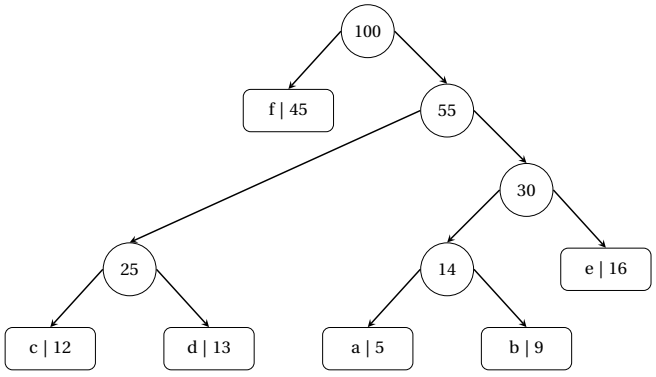
(b) La pila mínima contiene cuatro nodos, donde dos nodos son raíces de árboles con un solo elemento cada uno, y dos nodos de pila son raíces de árboles con más de un nodo.



(c) La pila mínima contiene tres nodos.



(d) La pila mínima contiene dos nodos.



(e) La pila mínima contiene un solo nodo.

Figura 8: Dada una tabla de probabilidades, se puede construir un «árbol de Huffman» para codificar cada objeto. Los pasos 2 y 3 del algoritmo se repiten hasta que todos los objetos estén combinados.

El algoritmo de Huffman crea un árbol binario a partir de los dos objetos con las probabilidades más bajas que se seleccionaron al inicio. Para ello, se etiqueta una rama con un «1» y la otra con un «0» (figura 9). No importa de qué lado se marque, siempre que el etiquetado sea consistente a lo largo del problema; es decir, el lado izquierdo siempre debe ser 1 y el lado derecho siempre debe ser 0, o viceversa.

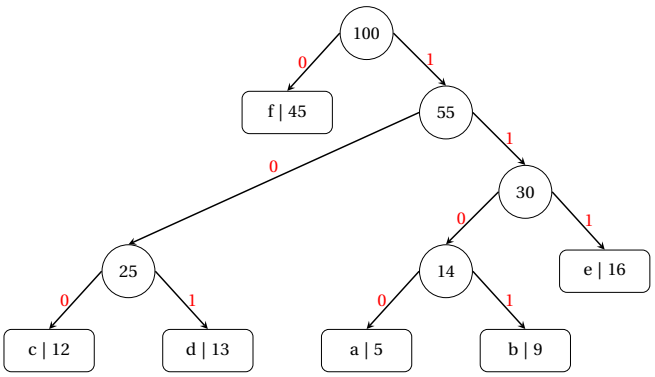


Figura 9: En un árbol binario de Huffman los nodos tienen como máximo dos hijos. Se puede distinguir entre ellos usando la dirección izquierda o derecha.

Por tanto, las codificaciones que se obtienen a partir del árbol son:

Carácter	f	c	d	a	b	e
Palabra de código	0	100	101	1100	1101	111

3.3.2. Algoritmo de Dijkstra

Un algoritmo para hallar la ruta más corta desde un nodo inicial hasta un nodo objetivo en un gráfico ponderado es el algoritmo de Dijkstra. El algoritmo crea un árbol de rutas más cortas desde el vértice inicial, la fuente, hasta hasta el resto de los puntos del gráfico [37, 38].

Supongamos que un estudiante quiere ir de su casa a la escuela de la manera más corta posible. Sabe que algunas carreteras están muy congestionadas y son difíciles de usar. En el algoritmo de Dijkstra, esto significa que el borde tiene un gran peso: el árbol de ruta más corto que encuentre el algoritmo intentará evitar los bordes con pesos más grandes. Si el estudiante busca direcciones usando un servicio de mapas, es probable que use el algoritmo de Dijkstra (figura 10).

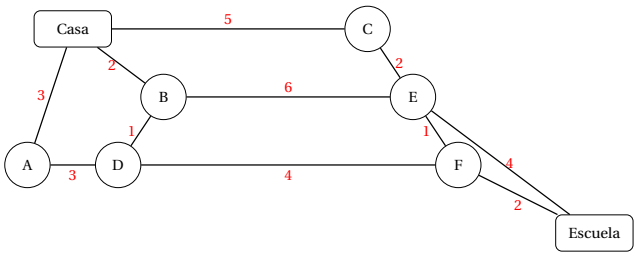


Figura 10: Gráfico ponderado que representa todos los caminos que existen entre la casa y la escuela.

El camino más corto de la figura 10, que se puede encontrar usando el algoritmo de Dijkstra, es:

$$\text{Casa} \to B \to D \to F \to \text{Escuela}.$$

Un algoritmo voraz para hallar el camino más corto entre dos puntos puede ser el siguiente: [39, 40]

1. Crea un conjunto de árboles que realice un seguimiento de los vértices incluidos en el árbol de ruta más corta, es decir, cuya distancia mínima desde la fuente se calcula y finaliza.
2. Asigna un valor de distancia a todos los vértices en el gráfico de entrada. Inicializa todos los valores de distancia como infinitos. Asigna el valor de distancia como 0 para el vértice de origen para que se elija primero.
3. Mientras que el conjunto de árboles no incluya todos los vértices:
   - Elige un vértice (u) que no esté en el conjunto y tenga un valor de distancia mínimo.
   - Incluye a u en el conjunto.
   - Actualiza el valor de la distancia de todos los vértices adyacentes de u, iterando a través de todos los vértices adyacentes. Para cada vértice adyacente v, si la suma del valor de la distancia de u (desde la fuente) y el peso del borde $u - v$ es menor que el valor de la distancia de v, actualiza el valor de la distancia de v.

Estudiemos el algoritmo con un ejemplo. Construyamos un gráfico ponderado (figura 11), elijamos un vértice inicial y asignemos valores de ruta infinitos al resto de los nodos.

- La distancia desde el nodo de origen a sí mismo siempre es 0 (figura 11a).
- La distancia desde el nodo de origen al resto de los nodos aún no se ha determinado. Esto se representa con el símbolo de infinito (∞).

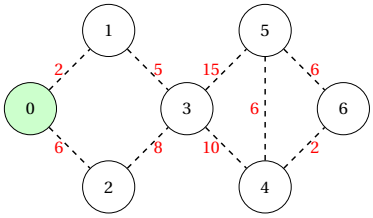
Nodo	0	1	2	3	4	5	6
Distancia	0	∞	∞	∞	∞	∞	∞

Necesitamos seleccionar el nodo más cercano al nodo de origen en función de las distancias conocidas, marcarlo como visitado (✓) e incluirlo a la ruta.

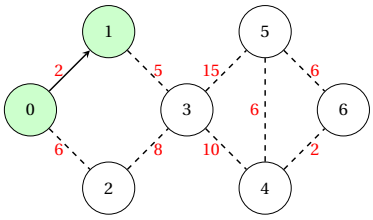
Si revisamos la lista de distancias, podemos ver que el nodo 1 tiene la distancia más corta al nodo de origen (una distancia de 2), por lo que lo agregamos a la ruta (figura 11b). Además, lo marcamos en la lista para indicar que ha sido visitado y que hallamos el camino más corto a este nodo.

Nodo	0	1	2	3	4	5	6
Distancia	0	2	∞	∞	∞	∞	∞
		✓					

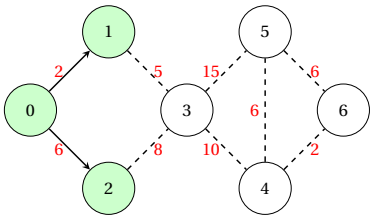
El nodo 2 y el nodo 3 son adyacentes a los nodos que ya están en la ruta, porque están conectados de forma directa al nodo 0 y al nodo 1, respectivamente. Estos son los nodos que analizaremos.



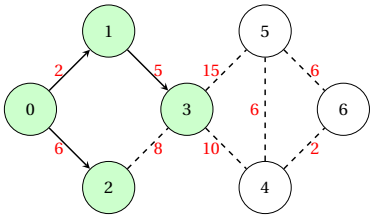
(a) El nodo de origen será el nodo 0, pero puede ser cualquier nodo.



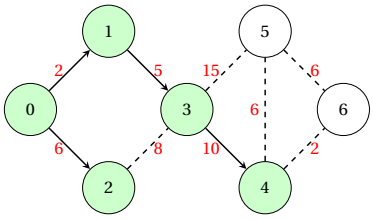
(b) El nodo 1 tiene la distancia más corta al nodo de origen.



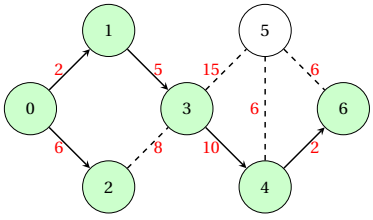
(c) El nodo 2 tiene la segunda distancia más corta al nodo de origen.



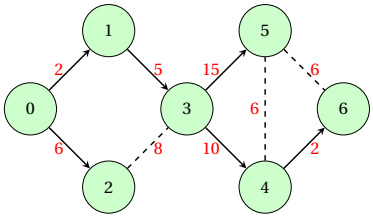
(d) La ruta 0 → 1 → 3 es la más corta entre los nodos 0 y 3.



(e) La ruta 0 → 1 → 3 → 4 es la más corta entre los nodos 0 y 4.



(f) La ruta 0 → 1 → 3 → 4 → 6 es la más corta entre los nodos 0 y 6.



(g) La ruta 0 → 1 → 3 → 5 es la más corta entre los nodos 0 y 5.

Figura 11: El algoritmo de Dijkstra encuentra el camino más corto entre un nodo dado, que se denomina «nodo de origen», y el resto de los nodos en un gráfico.

Lo primero que haremos será elegir qué nodo se agregará a la ruta. Debemos seleccionar el nodo no visitado con la distancia más corta (actualmente conocida) al nodo de origen. De la lista de distancias, podemos detectar de inmediato que este es el nodo 2 con la distancia 6 (figura 11c). Por tanto, marcamos el nodo 2 como visitado.

Nodo	0	1	2	3	4	5	6
Distancia	0	2	6	∞	∞	∞	∞
		✓	✓				

Para encontrar la distancia desde el nodo de origen a otro nodo (en este caso, el nodo 3), sumamos las longitudes de todas las aristas que forman el camino más corto para llegar a ese nodo.

- Para el nodo 3:

  La distancia total es 7 porque sumamos las longitudes de las aristas que forman el camino 0 → 1 → 3 (2 para la arista 0 → 1 y 5 para la arista 1 → 3).

  Además de la ruta 0 → 1 → 3, el camino 0 → 2 → 3 es otra opción que puede ser viable.

Veamos cómo decidir cuál de ellas es la ruta más corta. Si elegimos seguir el camino 0 → 2 → 3, necesitaríamos seguir dos aristas: 0 → 2 y 2 → 3 con longitudes 6 y 8, respectivamente, lo que representa una distancia total de 14.

Es claro que la primera distancia es más corta (7 frente a 14), por lo que elegiremos mantener la ruta 0 → 1 → 3 (figura 11d). Por tanto, marcamos el nodo 3 como visitado.

Nodo	0	1	2	3	4	5	6
Distancia	0	2	6	7	∞	∞	∞
		✓	✓	✓			

Ahora, necesitamos verificar los nuevos nodos adyacentes. Esta vez, son el nodo 4 y el nodo 5 ya que son adyacentes al nodo 3. Actualizamos las distancias de estos nodos al nodo de origen, siempre intentando encontrar el camino más corto:

- Para el nodo 4:

  La distancia es 2 + 5 + 10 = 17 desde 0 → 1 → 3 → 4.

- Para el nodo 5:

  La distancia es 2 + 5 + 15 = 22 desde 0 → 1 → 3 → 5.

Necesitamos elegir qué nodo no visitado se marcará como visitado. En este caso, es el nodo 4 porque tiene la distancia más corta en la lista de distancias (figura 11e).

Nodo	0	1	2	3	4	5	6
Distancia	0	2	6	7	17	22	∞
		✓	✓	✓	✓		

A continuación, comprobamos los nodos adyacentes: nodo 5 y nodo 6.

- Para el nodo 5:

  La primera opción es seguir la ruta $0 \rightarrow 1 \rightarrow 3 \rightarrow 5$, que tiene una distancia de 22 desde el nodo fuente. Esta distancia ya estaba registrada en la lista de distancias en un paso anterior.

  La segunda opción es seguir la ruta $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 5$, que tiene una distancia de $2+5+10+6=23$ desde el nodo fuente.

- Para el nodo 6:

  La ruta disponible es $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 6$, que tiene una distancia de $2+5+10+2=19$ desde el nodo fuente.

Marcamos el nodo con la distancia más corta (actualmente conocida) como visitado (figura 11f). En este caso, el nodo 6.

Nodo	0	1	2	3	4	5	6
Distancia	0	2	6	7	17	22	19
		✓	✓	✓	✓		✓

Solo un nodo no ha sido visitado todavía, el nodo 5. Veamos cómo podemos incluirlo en la ruta. Hay tres caminos diferentes que podemos tomar para llegar al nodo 5 de los nodos que se han agregado al camino:

- $0 \rightarrow 1 \rightarrow 3 \rightarrow 5$ con una distancia de $2+5+15=22$.
- $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ con una distancia de $2+5+10+6=23$.
- $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5$ con una distancia de $2+5+10+2+6=25$.

Es claro que la ruta más corta al nodo 5 es $0 \rightarrow 1 \rightarrow 3 \rightarrow 5$ (figura 11g). Marcamos el nodo como visitado de la lista de nodos no visitados.

Nodo	0	1	2	3	4	5	6
Distancia	0	2	6	7	17	22	19
		✓	✓	✓	✓	✓	✓

Tenemos el resultado final con el camino más corto desde el nodo 0 hasta cada nodo del gráfico.

3.4. Tiempo de ejecución

Pareciera que medir el tiempo de ejecución de un programa es algo fácil. Podríamos usar un cronómetro, iniciar el programa y observar cuánto tiempo pasa hasta que finaliza el programa. Pero este tipo de medida, conocido como tiempo de reloj de pared, no es, por varias razones, la mejor caracterización de un algoritmo informático [41].

Un punto de referencia describe solo el rendimiento de un programa en una máquina específica en un día determinado.Sin embargo, distintos procesadores pueden tener actuaciones dramáticamente diferentes. Incluso trabajando en la misma máquina, puede haber una selección de muchos compiladores alternativos para el mismo lenguaje de programación. Por esta y otras razones, los informáticos prefieren medir el tiempo de ejecución de una manera más abstracta [42].

3.4.1. Complejidad de algoritmos

La complejidad algorítmica es una medida de cuánto tiempo tardaría en completarse un algoritmo dada una entrada de tamaño n. Si un algoritmo tiene que escalar, debe calcular el resultado dentro de un límite de tiempo finito y práctico incluso para valores grandes de n. Por esta razón, la complejidad se calcula de forma asintótica cuando n tiende a infinito [43].

Si bien, la complejidad se expresa en términos de tiempo, a veces la complejidad también se analiza en términos de espacio, lo que se traduce en los requisitos de memoria del algoritmo [44].

3.4.2. Complejidad del tiempo

La complejidad del tiempo es la cantidad de tiempo que tarda un algoritmo en ejecutarse, en función de la longitud de la entrada. Mide el tiempo necesario para ejecutar cada declaración de código en un algoritmo. Aquí, la longitud de entrada indica el número de operaciones a realizar por el algoritmo [45].

En general, la complejidad del tiempo mide el tiempo necesario para ejecutar cada instrucción de código en un algoritmo. Si una declaración está configurada para ejecutarse repetidamente, la cantidad de veces que se ejecuta esa declaración es igual a n multiplicado por el tiempo requerido para ejecutar esa función cada vez [46].

3.4.3. Notación *Big O*

Existe una notación estándar que se usa para simplificar la comparación entre dos o más algoritmos: la notación *Big O*. La idea de esta notación es similar al concepto de la derivada en cálculo. Representa la tasa de crecimiento del tiempo de ejecución a medida que aumenta el número de elementos.

Decir que un algoritmo es $O(n)$ significa que el tiempo de ejecución está limitado por algunos tiempos constantes n. Esto se puede escribir como $c*n$. Si el tamaño del conjunto se duplica, entonces el tiempo de ejecución es $c*(2n)$. Pero esto es $2*(c*n)$, por lo que espera que el tiempo de ejecución también se duplique. Por otro lado, si un algoritmo de búsqueda es $O(\log n)$ y se duplica el tamaño del conjunto, se pasa de $c*(\log n)$ a $c*(\log 2n)$, que es simplemente $c+c*\log n$. Esto significa que los algoritmos $O(\log n)$ son más rápidos que los algoritmos $O(n)$, y esta diferencia solo aumenta a medida que aumenta el valor de n [47].

Una tarea que requiere la misma cantidad de tiempo, sin importar el tamaño de la entrada, se describe como $O(1)$ o «tiempo constante». Una tarea que es $O(n)$ se denomina tarea de «tiempo lineal». Una que es $O(log n)$ se llama logarítmica. Otros términos que se utilizan incluyen cuadrático para tareas $O(n^2)$ y cúbico para algoritmos $O(n^3)$ [48].

La tabla 2 muestra la complejidad del tiempo, en términos de la notación *Big O*, de algunos algoritmos simples.

Una limitación de la notación *Big O* es que no considera el uso de la memoria. Hay algoritmos que son teóricamente rápidos, pero usan una cantidad de memoria excesiva [49]. En situaciones como esta, puede ser preferible en la práctica un algoritmo más lento que use menos memoria.

Tabla 2: Complejidad del tiempo de algunos algoritmos de búsqueda y ordenamiento.

Algoritmo	Complejidad de tiempo		
	Mejor	Promedio	Peor
Lineal[1]	$O(1)$	$O(n)$	$O(n)$
Binaria[1]	$O(1)$	$O(\log n)$	$O(\log n)$
Burbuja[2]	$O(n)$	$O(n^2)$	$O(n^2)$
Inserción[2]	$O(n)$	$O(n^2)$	$O(n^2)$

[1]Algoritmo de búsqueda. [2]Algoritmo de ordenamiento.

3.5. Conclusiones

Los dispositivos informáticos son omnipresentes en la sociedad moderna. En las últimas décadas, hemos pasado de un mundo en el que estos dispositivos eran prácticamente desconocidos a un mundo en el que miles de millones de personas los usamos para llevar a cabo diversas actividades. Incluso nos hemos hecho dependientes de ellos.

Gran parte de los algoritmos subyacentes que permiten que los dispositivos informáticos funcionen de forma óptima son los mismos que se describieron en este trabajo. En aplicaciones comerciales, computación científica, ingeniería y muchas otras áreas de investigación, los algoritmos que se estudiaron marcan la diferencia entre poder resolver problemas en el mundo moderno y no poder abordarlos.

Referencias

[1] M. Habib, C. McDiarmid, J. Ramírez-Alfonsín y B. Reed. *Probabilistic Methods for Algorithmic Discrete Mathematics*. 1.ª edición. Berlín: Springer, 1998.

[2] J. Lodder, D. Pengelley y D. Ranjan (2013, julio). Euclid's Algorithm for the Greatest Common Divisor [En línea]. Disponible en www.maa.org.

[3] Geeks for Geeks (2022, febrero 3). Euclidean algorithms (Basic and Extended) [En línea]. Disponible en: www.geeksforgeeks.org.

[4] J. P. Tremblay, J. M. DeDourek y V. J. Friesen. *Programming in ADA*. 1.ª edición. McGraw-Hill Education, 1990.

[5] A. Aho, J. Ullman y J. Hopcroft. *Data Structures and Algorithms*. 1.ª edición. Pearson, 1983.

[6] E. E. García y J. A. Solano. Guía práctica de estudio 04: pseudocódigo [En línea]. Disponible en www.odin.fi-b.unam.mx.

[7] M. A. Rodríguez. *Metodología de la programación: a través del pseudocódigo*. 1.ª edición. Madrid: McGraw-Hill, 1991.

[8] R. Sedgewick. *Algorithms in C, Part 5: Graph Algorithms*. 3.ª edición. Harlow: Pearson, 2001.

[9] R. Sedgewick y P. Flajolet. *An Introduction to the Analysis of Algorithms*. 2.ª edición. Pearson, 2013.

[10] T. Alfaro. Algoritmos de Búsqueda y Ordenamiento [En línea]. Disponible en: www.inf.utfsm.cl.

[11] J. V. Álvarez. Tema 06: algoritmos de búsqueda y ordenación [En línea]. Disponible en: www.infor.uva.es.

[12] Universidad Europea de Madrid. Análisis semántico: la tabla de símbolos [En línea]. Disponible en: www.cartagena99.com.

[13] S. Gálvez. Tabla de símbolos [En línea]. Disponible en: www.ocw.uma.es.

[14] P. A. Sznajdleder. *Algoritmos a fondo con implementaciones en C y en JAVA*. 1.ª edición. Alfaomega, 2012.

[15] Principios de Programación: algoritmos de búsqueda y ordenación [En línea]. Disponible en: www.fing.edu.uy.

[16] E. K. Sáenz. Guía práctica de estudio 04: algoritmos de búsqueda parte 1 [En línea]. Disponible en: www.odin.fi-b.unam.mx.

[17] R. E. Neapolitan y K. Naimipour. *Foundations of Algorithms Using C++ Pseudocode*. 3.ª edición. Massachusetts: Jones and Bartlett Publishers, 2004.

[18] S. S. Skiena. *The Algorithm Design Manual*. 2.ª edición. New York: Springer, 2009.

[19] Algoritmos de ordenación y búsqueda [En línea]. Disponible en: www.mheducation.es.

[20] R. Johnsonbaugh. *Matemáticas discretas*. 4.ª edición. México: Pearson, 1997.

[21] V. M. de la Cueva, L. H. González y E. G. Salinas. *Estructuras de datos y algoritmos fundamentales*. 1.ª edición. México: Editorial Digital, 2020.

[22] J. Knowles. Sorting Algorithms: Correcness, Complexity and other Properties [En línea]. Disponible en: www.syllabus.cs.manchester.ac.uk.

[23] B. A. Cipra, «The Best of the 20th Century: Editors Name Top 10 Algorithms», SIAM *News*, vol. 33, no. 4, pp. 1-2, mayo 2000.

[24] M. Alberto, I. Schwer, V. Cámara y Y. Fumero *Matemática Discreta: con aplicaciones a las ciencias de la programación y de la computación*. 1.ª edición. Santa Fe: Ediciones UNL, 2005.

[25] M. T. Goodrich, R. Tamassia y D. M. Mount. *Data Structures and Algorithms in C++*. 2.ª edición. Estados Unidos de America: John Wiley & Sons, 2011.

[26] Algoritmos de Ordenamiento [En línea]. Disponible en: www.cs.buap.mx.

[27] L. Hernández. Fundamentos de la programación: algoritmos de ordenación [En línea]. Disponible en: www.fdi.ucm.es.

[28] A. Garrido. *Fundamentos de Programación en C++*. 1.ª edición. Madrid: Delta, 2006.

[29] G. Martín y F. A. Martínez. *Introducción a la programación estructurada en C*. 1.ª edición. Valencia: Quiles, 2003.

[30] Geeks for Geeks (2021, julio 8). Insertion Sort [En línea]. Disponible en: www.geeksforgeeks.org.

[31] N. Ziviani. *Diseño de algoritmos con implementaciones en Pascal y C.* 1. ª edición. Madrid: Thomson, 2007.

[32] J. Campos. Esquemas algorítmicos: algoritmos voraces [En línea]. Disponible en: www.webdiis.unizar.es.

[33] Geeks for Geeks (2022, abril 22). Greedy Algorithms [En línea]. Disponible en: www.geeksforgeeks.org.

[34] C. A. Shaffer. *Data Structures & Algorithm Analysis in C++*. 3.ª edición. Mineola: Dover Publications, 2011.

[35] Geeks for Geeks (2022, abril 19). Huffman Coding [En línea]. Disponible en: www.geeksforgeeks.org.

[36] Department of Mathematics and Computer Science. Huffman Coding [En línea]. Disponible en: www.math.oxford.emory.edu.

[37] M. McMillan. *Data Structures and Algorithms Using C#.* 1.ª edición. Cambridge, 2007.

[38] Geeks for Geeks (2022, febrero 22). Dijkstra's shortest path algorithm [En línea]. Disponible en: www.geeksforgeeks.org.

[39] S. Kadry, A. Abdallah y C. Joumaa. On The Optimization of Dijkstra's Algorithm [En línea]. Disponible en: www.arxiv.org.

[40] Favtutor. Dijkstra's Algorithm in C++: shortest Path Algorithm [En línea]. Disponible en: www.favtutor.com.

[41] Geeks for Geeks (2022, febrero 14). Measure execution time of a function in C++ [En línea]. Disponible en: www.geeksforgeeks.org.

[42] Laboratorio de Matemáticas. Tiempo de ejecución y eficiencia de algoritmos [En línea]. Disponible en: www.matematicas.uam.es.

[43] J. A. Mañas. Análisis de Algoritmos: complejidad [En línea]. Disponible en: www.dit.upm.es.

[44] Javatpoint. Complexity of Algorithm [En línea]. Disponible en: www.javatpoint.com.

[45] Geeks for Geeks (2022, marzo 2). Time Complexity and Space Complexity [En línea]. Disponible en: www.geeksforgeeks.org.

[46] Geeks for Geeks (2022, abril 28). Understanding Time Complexity with Simple Examples [En línea]. Disponible en: www.geeksforgeeks.org.

[47] N. M. Josuttis. *The C++ Standard Library: a tutorial and reference.* 2.ª edición. Estados Unidos de América: Pearson, 2012.

[48] M. Gregoire. *Professional C++*. 3.ª edición. Estados Unidos de América: John Wiley & Sons, 2014.

[49] Geeks for Geeks (2021, marzo 6). Analysis of Algorithms: big-O analysis [En línea]. Disponible en: www.geeksforgeeks.org.